CS 315 - Programming Languages
Project 1 Report

Language name: Rudelang

Can Ersoy - 22003216 - Section 2
Tolga Özgün - 22003850 - Section 1
Selim Can Güler - 22002811 - Section 1

Table Of Contents

1) The complete BNF description

a) Program definition and statements

<program> ::= <start_function> | <start_function><function_list>

<start_function> ::= <data_type> start(){<statement_list>}

<function_list> ::= <function> | <function_list><function>

<function> ::= <return_type> func <identifier> (<parameter_list>)
{<statement_list>}

<statement_list> ::= <statement> | <statement_list> <statement>

<parameter_list> ::= <parameter> | <parameter_list>, <parameter>

<parameter> ::= <data_type> <identifier>

<statement> ::= <while_statement> | <for_statement>
| <definition_statement> | <if_statement> | <return_statement>
| <comment_statement> | <expression_statement>

<definition_statement> ::= <declaration_statement> | <assignment_statement>
| <initialization_statement> | <empty_statement>

<declaration_statement> ::= <data_type> <identifier>;

<assignment_statement> ::= <identifier> = <expression>;

<initialization_statement> ::= <data_type> <identifier> = <expression>;

<empty_statement> ::= ;

<data_type>::= int | long | float | double | boolean | string | char

<return_type> ::= void | <data_type>

<expression_statement> ::= <expression>;

<return_statement> ::= return <identifier>; | <number>; | <string_literal>;
| <boolean_literal>; | <character_literal>; | ;

<comment_statement > ::= /# <string_literal> #/

b) Expressions

<expression> ::= <conditional_expression>

<conditional_expression>::= <conditional_and>
| <conditional_expression> <logical_or> <conditional_and>

<conditional_and>::= <equality_expr>
| <conditional_and> <logical_and> <equality_expr>

<equality_expr>::= <relational_expr>
| <equality_expr> == <relational_expr>
| <equality_expr> != <relational_expr>

<relational_expr>::= <arithmetic_add_expr>
| <relational_expr> < <arithmetic_add_expr>
| <relational_expr> > <arithmetic_add_expr>
| <relational_expr> <= <arithmetic_add_expr>
| <relational_expr> >= <arithmetic_add_expr>

<arithmetic_add_expr>::= <arithmetic_mult_expr>
| <arithmetic_add_expr> + <arithmetic_mult_expr>
| <arithmetic_add_expr> - <arithmetic_mult_expr>

<arithmetic_mult_expr>::= <arithmetic_base>
| <arithmetic_mult_expr> * <arithmetic_base>
| <arithmetic_mult_expr> / <arithmetic_base>
| <arithmetic_mult_expr> ** <arithmetic_base>

<arithmetic_base>::= <boolean_literal>
| <number>
| <string_literal>
| <char_literal>
| (<expression>)
| <identifier>
| <function_call>

c) Conditional statements

<if_statement> ::= if(<conditional_expression>){<statement_list>}
<else_if_statements><else_statement>

<else_if_statements> ::= <empty>
| elif(<conditional_expression>){<statement_list>}<else_if_statements>

<else_statement> ::= <empty> | else{<statement_list>}

d) Loops

<while_statement> ::= while (<conditional_expression>){<statement_list>}

<for_statement> ::= for(<definition_statement> <conditional_expression>;
<definition_statement>) {<statement_list>}

e) Built-in functions

<builtin_function> ::= getCurrentTimestamp()
| connectToURL(<string_literal> | <identifier> | <function_call>)
| getDataFromURL(<string_literal> | <identifier>)
| sendDataToURL(<string_literal> | <identifier>,<integer> | <identifier>)
| connectToInternet()
| isConnectedToInternet()
| print(<string_literal> | <number>| <identifier>)
| enter()
| toggleSwitchOn(<digit>)
| toggleSwitchOff(<digit>)
| readSensorData(<sensor_type>)
| readSensorData(<sensor_type>, <number>)

f) Function call

<function_call> ::= pls <function_name>(<argument_list>)
| pls <builtin_function>

<argument_list>::= <argument_list>,<returnable>
| <returnable>
| /* empty argument */

<returnable> ::= <identifier> | <number> | <function_call> | <literal>

g) Terminals

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<lowercase_letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<uppercase_letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<symbol> ::= ! | @ | # | $ | % | ^ | & | * | (|) | - | _ | + | = | ` | ~ | < | > | , | . | ; | : | “ | | | \ |

<comparison_op> ::= < | > | <= | >= | == | !=

<sensor_type> ::= temperature | humidity | air_pressure | air_quality | light | sound_level

<logical_expression> ::= <logical_or> | <logical_and>
<logical_or>::= ||
<logical_and>::= &&

h) Alphabetics, literals, numbers

<letter> ::= <lowercase_letter> | <uppercase_letter>

<alphabetic> ::= <letter>
| <alphabetic> <letter>

<alphanumeric> ::= <alphabetic>
| <digit>
| <alphanumeric> <digit>
| <alphanumeric> <alphabetic>

<identifier> ::= <alphabetic>
| <letter> <alphanumeric>

<literal> ::= <boolean_literal> | <string_literal> | <char_literal>

<boolean_literal> ::= true | false

<string_literal> ::= “<string>”

<string> ::= <empty> | <character_literal> | <string> <character_literal>

<empty> ::=

<character_literal> ::= <letter> | <digit> | <symbol>

<integer> ::= <digit>
| <integer> <digit>

<number> ::= <integer>
| <integer> . <integer>
| .<integer>

2) Documentation

<program> ::= <start_function> | <start_function><function_list>

This is the official definition of the program. The program will consist of either only the start function or it will be a combination of the start function and other functions defined by the user.

<start_function> ::= <data_type> start(){<statement_list>}

Start function works like a regular function definition. However, its name is reserved and called “start”. Start function by design must be defined for the program to work. The function will contain a statement list. The program will work even though the statement list is empty.

By a design choice, the definition of the start function is different from those of regular function definitions. The start function does not need the “func” keyword in its definition. Whereas a regular function definition requires the use of “func”.

<function_list> ::= <function> | <function><function_list>

A function list is a list of function definitions. These are functions outside the scope of the start function. The user will define these functions, and unlike the start function, these functions are optional and there for the user's convenience.

<function> ::= <return_type> func <identifier> (<parameter_list>) {<statement_list>}

A function definition will include the return type of the function, the “func” reserved word, the name of the function, a list of parameters, and a statement list. The function's return type includes “void” apart from the regular data types defined in the language.

Example:
```
float func multiply(float firstNum, float secondNum) {
        return (firstNum * secondNum);
}
```

<parameter_list> ::= <parameter> | <parameter_list>, <parameter>

A parameter list consists of only one or multiple parameters. A comma must separate each parameter.

<parameter> ::= <data_type> <identifier>

A data type and an identifier define a parameter.

Example: int noOfLights

<statement_list> ::= <statement> | <statement_list> <statement>

There may be only one or multiple statements inside a statement list.

<statement> ::= <while_statement> | <for_statement>
| <definition_statement> | <if_statement> | <return_statement>
| <comment_statement> | <expression_statement>

A statement can be a while statement, for statement, declaration statement, assignment statement, initialization statement, conditional if statement, return statement, comment statement, expression statement, or it may just be empty. Each statement must be ended with the symbol ";"

<definition_statement> ::= <declaration_statement>
| <assignment_statement> | <initialization_statement>
| <empty_statement>

A definition statement is a general construct that declares a variable, or assigns a value to an already declared variable, or does the declaration and assignment simultaneously.

<declaration_statement> ::= <data_type> <identifier>;

A declaration statement creates a variable without assigning it any value. The declared variable may be used later.

<assignment_statement> ::= <identifier> = <expression>;

An assignment statement assigns a value to an already declared variable. The variable should be declared before the assignment statement for this statement to work.

<initialization_statement> ::= <data_type> <identifier> = <expression>;

An initialization statement declares a variable and assigns a value to the declared variable simultaneously.

<empty_statement> ::= ;

A statement can be empty.

<data_type>::= int | float | double | boolean | string | long | char

We have seven data types in Rudelang, int represents whole numbers, and long is for larger whole numbers. float and double are decimal point numbers. boolean is either true or false. a string is just concatenated characters. char is a single character.

<return_type> ::= void | <data_type>

The return type is used for the return types in the function definitions. Return type is either a regular data type or void since a function may not return anything. A void reserved word simply represents the non-existence of a return statement.

<return_statement> ::= return <identifier>; | return <number>;
| return <string_literal>; | return <boolean_literal>;
| return <character_literal>;
| return;

A return statement marks the end of a function. The return statement may return an already defined identifier, or it can simply return a number, a string literal, a boolean literal, or a character literal. The return statement may be empty but should always end with a semicolon.

<comment_statement > ::= /# <string_literal> #/

Comments are opened by "/#" and closed by "#/".

<expression> ::= <conditional_expression>

Expressions are listed under conditional expressions as we wanted to manage the precedence between conditional and arithmetic expressions.

<conditional_expression>::= <and_expressions>
| <expression> <or_expression> <and_expressions>

Conditional expressions can be either an "and expression", or an "expression" followed by an "or expression" and an "and expression". This declaration ensures that "or expressions" have precedence over "and expressions".

<conditional_and>::= <equality_expr>
| <conditional_and> <logical_and> <equality_expr>

"Conditional and expressions" are extended either as an "equality expression", or a "conditional and expression" followed by a "logical and expression" and an "equality expression".

<equality_expr>::= <relational_expr>
| <equality_expr> == <relational_expr>
| <equality_expr> != <relational_expr>

Equality expressions have three possible notations. They can either be a relational expression, or an equality expression with equals or not equals operator followed by relational expression.

<relational_expr>::= <arithmetic_add_expr>
| <relational_expr> < <arithmetic_add_expr>
| <relational_expr> > <arithmetic_add_expr>
| <relational_expr> <= <arithmetic_add_expr>
| <relational_expr> >= <arithmetic_add_expr>

Relational expressions are either bare arithmetic add expressions, or any relational expression followed by a relational expression and an arithmetic add expression.

<arithmetic_add_expr>::= <arithmetic_mult_expr>
| <arithmetic_add_expr> + <arithmetic_mult_expr>
| <arithmetic_add_expr> - <arithmetic_mult_expr>

Arithmetic addition expressions can either be just arithmetic multiplication expressions or addition expressions followed by plus or minus sign and arithmetic multiplication expressions.

<arithmetic_mult_expr>::= <arithmetic_base>
| <arithmetic_mult_expr> * <arithmetic_base>
| <arithmetic_mult_expr> / <arithmetic_base>
| <arithmetic_mult_expr> ** <arithmetic_base>

Arithmetic multiplication expressions are either arithmetic bases or arithmetic multiplication expressions followed by multiplication, division or factor (** symbol stands for the power symbol) and arithmetic base.

<arithmetic_base>::= <boolean_literal>
| <number>
| <string_literal>
| <char_literal>
| (<expression>)
| <identifier>
| <function_call>

Arithmetic base can either be a boolean, a number, a string, a character, an expression, an identifier or a function call.

<if_statement> ::= if(<conditional_expression>){<statement_list>}
<else_if_statements><else_statement>

If statement checks the conditional expression and runs its statements if its condition holds. The if statement can have multiple else if statements or none. Else if conditionals are only checked if the conditionals before them did not meet. Else statement is optional and can only be used once. It runs if none of the conditions defined before were met.

<else_if_statements> ::= <empty>
| elif(<conditional_expression>){<statement_list>}<else_if_statements>

Else if statements may be empty or it may contain other else-if statements. Else if statements are run only if the previous if or else if statements' conditions are not met.

<else_statement> ::= <empty> | else{<statement_list>}

Else statements may be empty or they may contain statements. Else statements are run only if the previous if or else if statements' conditions are not met.

<while_statement> ::= while (<conditional_expression>){<statement_list>}

While statement is a kind of loop that continuously runs the statement list provided until the conditional_expression returns false.

<for_statement> ::= for(<definition_statement> <conditional_expression>; <definition_statement>) {<statement_list>}

For statement is a kind of loop. When it is first run, it runs the definition statement. After each iteration, the conditional expression is checked to determine whether the loop should end or not. Moreover, after each iteration, there is a statement as an update.

<function_call> ::= pls <function_name>(<param_list>)
| pls <builtin_function>

A function is called using the reserved word "pls" followed by the function identifier and the parameters list. "pls" stands for the word please.

<argument_list>::= <argument_list>,<returnable>
| <returnable>
| /* empty argument */

Argument list is defined in order to allow the user to give arguments to functions. The arguments can be a list of returnable arguments or an empty argument.

<returnable> ::= <identifier> | <number> | <function_call> | <literal>

A returnable should be written after 'return' keyword and it is the return value of the function.

<sensor_type> ::= TEMPERATURE | HUMIDITY | AIR_PRESSURE | AIR_QUALITY | LIGHT | SOUND_LEVEL

We defined special types of sensors so that it will be easier for the users of the language.

<logical_or>::= ||

The symbol || is the terminal for logical or operations.

<logical_and>::= &&

The symbol && is the terminal for “logical and” operations. The precedence for logical and operation and logical or operation is handled in a way that the program will handle logical and operations first.

3) Documentation of built-in functions

getCurrentTimestamp()

Returns long. It gets the current timestamp from the timer (in terms of seconds) and returns it.

connectToURL(<string_literal>)

Connects to the given URL if the URL is valid. Returns true if the connection is successful, else false.

getDataFromURL(<string_literal> | <identifier>)

Returns an integer from the already connected URL.

sendDataToURL(<string_literal> | <identifier>, <integer>)

The return type of this function is void. This function sends given integer data to the already connected URL.

connectToInternet()

Connects to the internet using the internet adapter plugged into the hardware. The connection credentials are assumed to be encoded inside the hardware for security. Returns true if the connection is successful, else false.

isConnectedToInternet()

Checks if the IoT node is connected to the internet at any moment. If it is connected, returns true, and otherwise, false.

print(<string_literal> | <number>):

Prints the given string literal or number into the console, the main screen of the IoT device.

enter():

This function allows the user to enter an input. It returns the entered input.

toggleSwitchOn(<digit>):

Toggles the specific switch on whose id number is given as a parameter where the digit is an integer between 0 and 9 inclusive.

toggleSwitchOff(<digit>):

Toggles the specific switch off whose id number is given as a parameter where the digit is an integer between 0 and 9.

readSensorData(<sensor_type>):

Reads sensor data at the specific time of the call of this function from the specified sensor type in the parameter.

readSensorData(<sensor_type>, <number>):

Reads sensor data from the sound level sensor at the specific time of the call of this function from the specified sensor type in the parameter.

4) Description of non-trivial tokens

Comments

Comment blocks are opened with /# and closed with #/. This definition can work both as a single-line comment and a multi-line comment. Anything between the opening and closing comment symbols is ignored by the language and assumed to be there for the sake of readability.

Identifiers

Identifiers must start with an alphabetic character and may continue with alphanumeric characters.

Identifiers follow the camel-case naming convention. They represent variable or function names.

Literals

We have three types of literals: boolean, character, and string. Boolean only accepts true or false. Character accepts a letter, a digit, or a basic ASCII symbol. A string can be empty or a sequence of characters. We accepted empty strings as this project is focused on IoT devices, which may return results with empty strings, where we do not want to display or transmit any kind of data.

Reserved Words

for: Reserved word for initializing for statements.
while: Reserved word for initializing while statements.
if: Reserved word for initializing if statements.
else: Reserved word for initializing else statements.
elif: Reserved word for initializing else if statements.
int: Reserved word for integer data type.
long: Reserved word for long data type (2^32 times larger than integer).
float: Reserved word for floating point data type.
double: Reserved word for double data type.
boolean: Reserved word for boolean data type.
string: Reserved word for string data type.
char: Reserved word for character data type.
void: Reserved word for void return type.
pls: Reserved word for calling functions. Stands for “please”.
endpls: Reserved word for ending the program file. Stands for “end please”.
return: Reserved word for initializing return statements.
true: Reserved word for true boolean literal.
false: Reserved word for false boolean literal.
endpls: Reserved word for end of file input.
TEMPERATURE: Reserved word for temperature sensor type.
HUMIDITY: Reserved word for humidity sensor type.
AIR_PRESSURE: Reserved word for air pressure sensor type.
AIR_QUALITY: Reserved word for air quality sensor type.
LIGHT: Reserved word for light sensor type.
SOUND_LEVEL: Reserved word for sound level sensor type.
getCurrentTimestamp: Reserved word for the built-in function to get current timestamp.
connectToURL: Reserved word for the built-in function to connect to a given URL.
getDataFromURL: Reserved word for the built-in function to connect to get data from a given URL.
sendDataToURL: Reserved word for the built-in function to send data to a given URL.
connectToInternet: Reserved word for the built-in function to connect to the internet
isConnectedToInternet: Reserved word for the built-in function to check whether the IoT node is connected to the internet.
print: Reserved word for the built-in function to print a value to the console.
enter: Reserved word for the built-in function to wait for user input.
toggleSwitchOn: Reserved word for the built-in function to toggle a switch on.
toggleSwitchOff: Reserved word for the built-in function to toggle a switch off.
readSensorData: Reserved word for the built-in function to read data from a given sensor.

For generic constructs like loops, conditional statements, and data types, we chose words that are similar to words of popular programming languages.

To call a function, we required the users to use the word “pls” (please) because we wanted the language to be more readable. Since the functions conventionally start with verbs, function calls are more readable and reliable with “pls” as if we ask the compiler to do something for us kindly.

Rude is a special-purpose programming language to program IoT nodes. We specified reserved words for different sensors to make it easier for the programmers to access sensors. This increases the writability along with the readability of the program.

5) Evaluation in terms of readability, writability, reliability

Readability

In Rudelang, we did not allow symbols while naming identifiers. We recognized that C-type programs implemented a great rule of not allowing symbols except for underscore (_) and dollar sign ($). However, we thought this rule could be improved by not allowing any symbols while naming identifiers. To elaborate, we recognized that the conventional use of the underscore (_) symbol is to distinguish private and public identifiers. As Rudelang does not involve public and private keywords, we decided against allowing it. Also, the dollar sign is conventionally used for specifying class names declared under a class. Likewise, Rudelang does not introduce the notion of classes, and we decided that the dollar sign convention was not useful for our language.

Moreover, we introduced a new keyword, “pls” for function calls. Commonly used C-type programming languages, such as Java, C and C++, do not involve a keyword for function calls, and we believe this reduces the readability of the code. In these types of languages, a programmer may mistakenly read a function as a variable, or vice versa. However, in our language, a programmer can locate the function calls by tracking the use of the keyword “pls”.

On the contrary of some languages that do not implement curly brackets to define the scope of conditional statements and loops, we force the programmers to use curly brackets. Having only one consistent way of writing these language constructs increases readability by creating a consistent form for these statements. For example, in Java, if and for statements that have only one statement can be written without the use of curly brackets; however, this reduces readability. That is why we decided not to implement it to improve the readability of our own language.

The symbols for the comment block are chosen in a way that they will resemble an opening and closing pair, making the comments more readable.

We have a keyword that defines where the program ends. That keyword is called “endpls”. This keyword is introduced to increase readability by giving the programmer a pointer that points to where the program ends.

Writability

We support all combinations of conditional expressions in Rudelang, which gives the user of this programming language the ability to check for any number of conditions. For example, a user might want to read multiple sensor data and command the device to turn a switch on when a condition including all of these sensor data is met. To allow such operations, we allowed any kind of expression that returns a boolean value to be a part of the conditional expression. This includes boolean literals, variables that return true or false, functions that return true or false, and arithmetic comparisons. We can combine each of these via the use of logical operators. Also, having the option for multiline comments increases the writability of the comment block.

We provided various built-in functions, such as readSensorData(sensorType). We also have readSensorData(sensorType, frequency) to increase writability as we allow the users to specify the desired frequency for sound sensors. However, this causes a drop in reliability since if the user does not specify a frequency but still wants to read the sound sensor, they will be presented with a default frequency.

Reliability

We require using curly brackets in loops, conditional statements, and function definitions. Doing so removes the confusion around one-line loops, conditional statements, and functions. If we were to let doing these in one line, the user of the language might have forgotten to place curly brackets. In such a case, only the line after the statement would be executed, and the program would produce unreliable results.

Since Rudelang is a special-purpose programming language to program IoT nodes, we created data types like long and double which can store larger numbers compared to int and float. This reduces the chance of overflow and unexpected results related to overflowing. Therefore, we increase the reliability by creating the same result on any kind of arithmetic expression.

Moreover, we added the keyword “pls” to introduce another level of reliability where the function calls would be preceded with a keyword, and the code functions as the programmer intended.

6) How Precedence is Handled

“Logical and” and “logical or” operations have a precedence relation between each other, where “logical and” is always handled before “logical or”. We handled this case by using layers in our BNF description and yacc file. Logical operations were described inside conditional expressions in the BNF.

Conditional expressions include logical expressions, relational expressions, and arithmetic expressions. We layered each of these expressions in a way to handle precedence. The first layer includes logical expressions. Logical expressions are divided into two layers. This allows us to give precedence to AND operation over OR operation.

The structure of arithmetic expressions is similar to logical expressions. The multiplication operation is given precedence over the addition operation by putting the

multiplication in a layer embedded within the addition operation. Also, the definition of arithmetic expressions is left-associative, which handles division and subtraction correctly.